Efficient Java Native Interface for Android based Mobile Devices

by

Preetham Chandrian

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved May 2011 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Hasan Davlcu
Baoxin Li

ARIZONA STATE UNIVERSITY

August 2011

ABSTRACT

Currently Java is making its way into the embedded systems and mobile devices like androids. The programs written in Java are compiled into machine independent binary class byte codes. A Java Virtual Machine (JVM) executes these classes. The Java platform additionally specifies the Java Native Interface (JNI). JNI allows Java code that runs within a JVM to interoperate with applications or libraries that are written in other languages and compiled to the host CPU ISA. JNI plays an important role in embedded system as it provides a mechanism to interact with libraries specific to the platform.

This thesis addresses the overhead incurred in the JNI due to reflection and serialization when objects are accessed on android based mobile devices. It provides techniques to reduce this overhead. It also provides an API to access objects through its reference through pinning its memory location. The Android emulator was used to evaluate the performance of these techniques and we observed that there was 5 - 10 % performance gain in the new Java Native Interface.

To My Beloved Family

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor; Dr. Yann-Hang Lee, without whose guidance, encouragement and support, this thesis would not have been possible. I have gained a lot of knowledge about my field, improved my research skills and otherwise, working under him. In all it has been a very satisfying and a fulfilling experience.

I would also like to thank Dr. Hasan Davulcu and Dr. Baoxin Li for their time and effort to help me fulfill my degree requirements; as part of my thesis committee.

I am grateful to all my friends at Arizona State University who made me feel at home and making this stay an enjoyable experience. I am thankful to all my lab mates, especially Yaksh Sharma, Zhou Fang and Rohit Girme who helped me with my queries.

Finally, I am indebted to my parents and my family. They all have constantly encouraged me and been understanding through all the tough times. I wish to express my utmost and deepest gratitude to them for being patient, warm and supportive.

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

With the introduction of JavaME (Micro Edition) by Sun MicroSystem for mobile and embedded devices Java has rapidly acquired a significant amount of market share. Also the applications based on JavaME are portable across devices developers favor Java over other programming languages. The problems with using Java are following:

- Standard Java libraries do not support platform specific features that are needed by the application.
- If the developers wants to use libraries written in different languages because of the fact that these libraries are more efficient and faster than its Java counterpart.
- If the developer wants to implement time critical code in lower level language.

Java Native Interface (JNI) addresses all these issues. It provides the following functions:

- It can create, update and inspect Java objects which include arrays.
- It can call Java methods from C/C++
- It can catch and raise exception from native code.
- Native code can load class and request class information.

JNI has played a major role in development of application which needs to interact

with native code (C/C++). In recent years Java has become one of the favorite application programming languages and is widely used by developers. This is evident in android as well as majority of the application are written in Java. But the fact that the operating system that android is developed on is Linux like which requires JNI to interact with the devices. As more and more features are being added to the mobile devices there is a need for an efficient JNI. There are papers [1] which demonstrate the overhead incurred while making JNI calls. It provides detailed performance benchmarks of several popular, modern, and representative JNI implementations, pointing out their weak points and suggesting possible solutions. There is a need to decrease these overhead in embedded systems like android phones. Foreign-function interfaces (FFIs) such as “Jeannie” have been developed to improve safety and productivity of JNI.

JNI are commonly used to interact with native libraries which involves processing of chunks of data. The data transfer between the JVM address space and that of the native address space is costly. Further the overhead occurred in reflection and serialization of not primitive data is significant.

1.1 Motivation

Android uses JNI in its NDK (Native Development Kit), which is a toolset that helps developers to interact with native code components in their application. The NDK provides the following:

- It provides tools and builds files that are helpful for developers to generate native code libraries from C and C++ sources.

- It helps developers embed native libraries into an application package file.
- A set of native system headers and libraries that will be supported in all future versions of the Android platform, starting from Android 1.5. Applications that use native activities must be run on Android 2.3 or later.

NDKs are used to build activity, handle user inputs, use hardware sensors and platform specific operations and the fact that they heavily rely on JNI. Making JNI faster will not only make the application response time faster but also reduces the use power consumption of the device. The bottle neck in JNI is the data transfer between the JVM memory space and the native memory space. There is also time consumed while accessing class structure and fieldIds of class attributes. The transfer of data can be reduced if we can pin the memory address of the object or array that we intend to manipulate. If we could cache the class structure and the fieldIds of its attributes the over head of searching the class structure can be reduced. As each call to the JNI is treated as new call without the information of its previous history a lot of information is lost. If this information were available to us we could use it and reduce the execution time of the JNI.
One of the approaches is the use of JIT (Just In Time) compilers to inline the JNI calls. This reduces the cost of callouts to native code by reducing the over head of stack operations that needs to be performed during the JNI calls.

The proposed solution alters the JNIEnv structure to include hash map to store the recently accessed fieldIds and methods. It also has caches the class structure in the JNIEnv structure. A new method is introduced to pin the address

of the Java object so that the same address can be used in future access this decreases the access time and also the information can be accessed without the use of JNI as we already know its location in the memory. To access such kind of address the program should be a thread in the same process as that of the JVM.

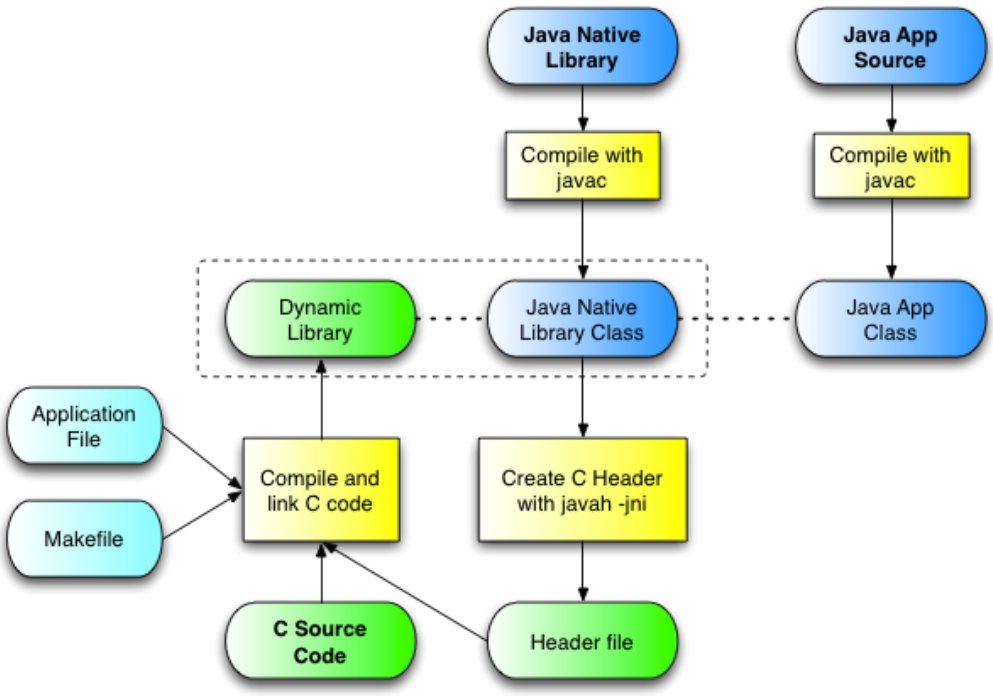


Figure 1: Overview of JNI

1.2 Document Outline

The rest of the document is organized as follows.

Chapter 2, 3 provides Background information about Android, its architecture, and JNI. It also explains some terms to better understand the document.

Chapter 4 talks about the Related Work done in the area of JNI in general. It also mentions certain works which describe the specific mechanism and techniques

used to make JNI developer friendly and efficient.

Chapter 5 talks about the design and techniques that are used to make Android JNI efficient as a whole. It mentions the detailed working of various JNI calls and way they are made efficient.

Chapter 6 describes the implementation details.

Chapter 7 gives a various measuring techniques used for comparing the efficiency of the new JNI aver the existing one. It concludes the thesis as a whole.

Chapter 8 presents some features and enhancements that can be done to the existing implementation. This will further enhance the tool.

CHAPTER 2

Java Native Interface (JNI)

2.1 Role of JNI

When the Java platform is deployed on top of host environments, it may become desirable or necessary to allow Java applications to work closely with native code written in other languages. Java is slowly replacing the programs that were written in C and C++ as they are platform independent. The Java Native Interface is a feature that allows the programmer to take advantage of the Java virtual machine, but still can use code written in other languages and libraries. As a part of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa. Figure 2 illustrates the role of the JNI.

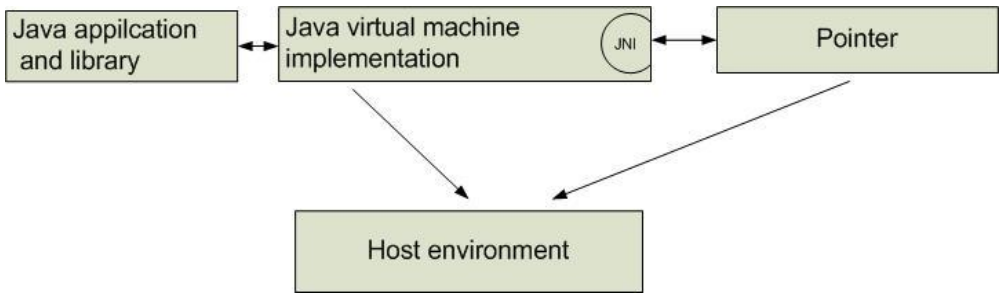


Figure 2: Role of JNI

The JNI is designed to handle situations where you need to combine Java applications with native code. It supports calls from native library and calls from native application.

- The JNI can be used to write native methods that allow Java applications to call functions implemented that are implemented in native libraries. Java applications call native methods in a similar way that is a simple Java method invocation only difference is that it has the keyword *native* in its prototype. But these Java calls are internally translated to calls to the native that call the native method or library.
- The JNI supports an invocation interface that allows us to call Java virtual machine code from the native code. Native applications can link with a native library that uses Java API, and then the native code can use the JNI APIs to call these functions in the Java virtual machine.

2.2 Overview

The figure below illustrates the steps required write a simple application that invokes a C function.

1. Create a class (HelloWorld.java) that declares the native method.

2. Use javac to compile the HelloWorld source file, resulting in the class file HelloWorld.class. The javac compiler is supplied with JDK or Java 2 SDK releases.

3. Use javah -jni to generate a C header file (HelloWorld.h) containing the function prototype for the native method implementation. The javah tool is provided with JDK or Java 2 SDK releases.

4. Write the C implementation (HelloWorld.c) of the native method.

5. Compile the C implementation into a native library, creating HelloWorld.dll or

libHelloWorld.so. Use the C compiler and linker available on the host.

Run the HelloWorld program using the Java runtime interpreter. Both the class files (HelloWorld.class) and the native library (HelloWorld.dll or libHelloWorld.so) are loaded at runtime.

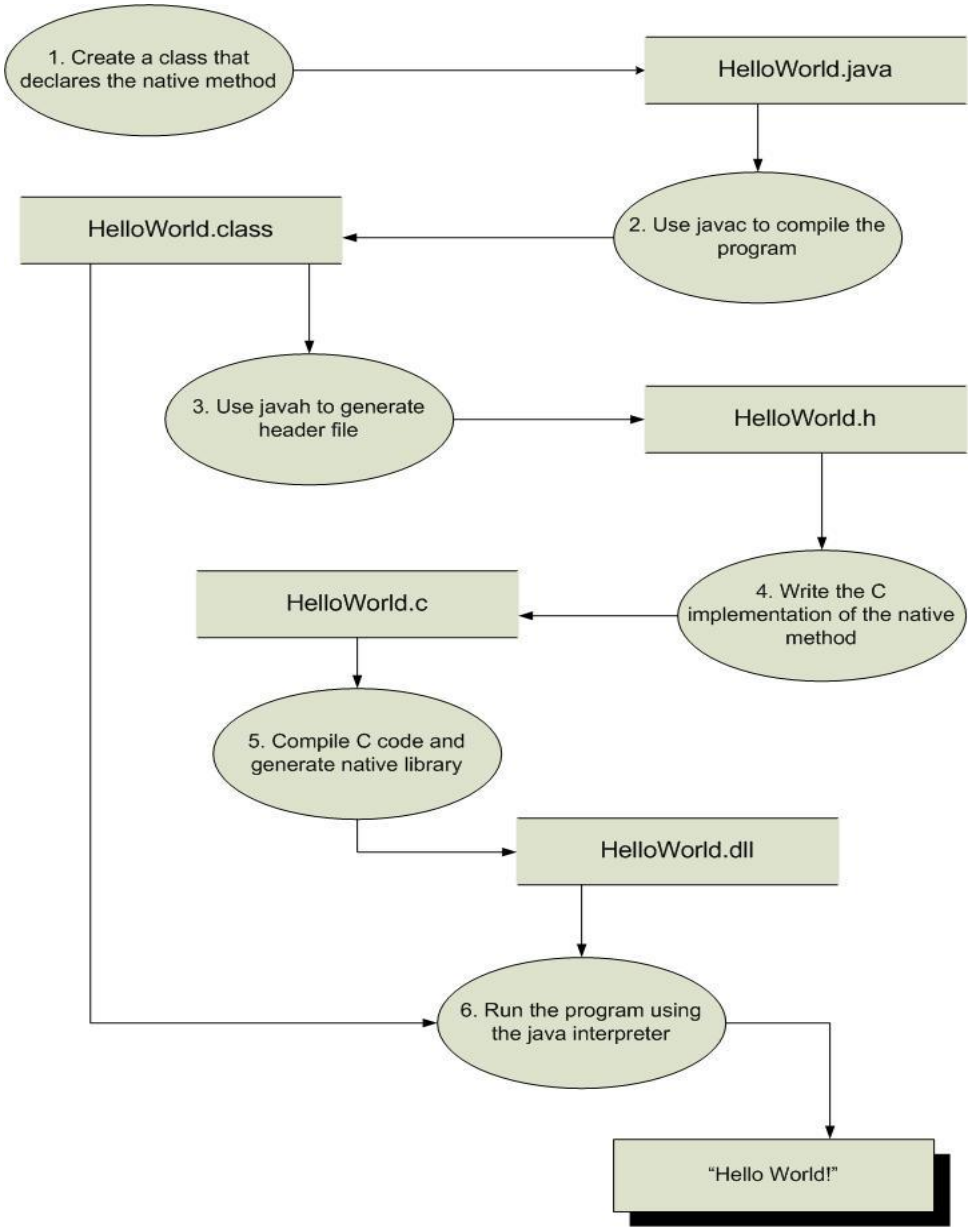


Figure 3: Steps writing a JNI program

The native method in Java is declared as follows

```
class HellWorld {
    private native void print();
        :
        :
}
```

2.3 Native Method Arguments

As discussed in the previous section, the native method implementation such as Java_Prompt_getLine accepts two standard parameters, in addition to the arguments declared in the native method. The first parameter, the JNIEnv interface pointer, points to a location that contains a pointer to a function table. Each entry in the function table points to a *JNI function*. Native methods always access data structures in the Java virtual machine through one of the JNI functions. Figure illustrates the JNIEnv interface pointer.

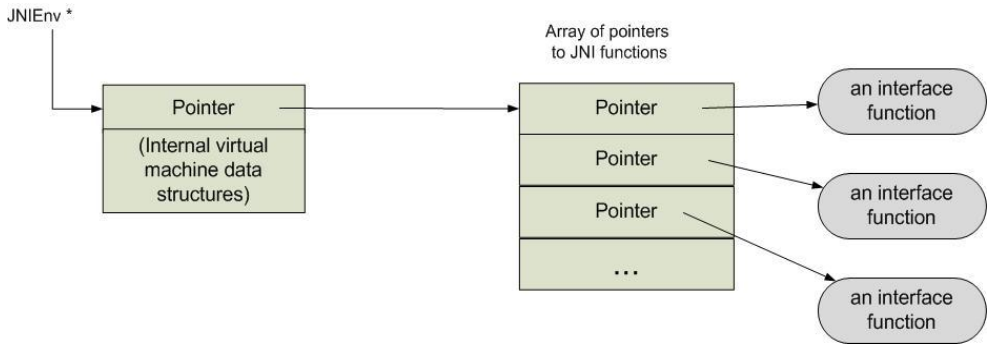


Figure 4: The JNIEnv interface pointer

The second argument differs depending on whether the native method is a static or an instance method. The second argument to an *instance* native method is a reference to the object on which the method is invoked, similar to this pointer in C++. The second argument to a *static* native method is a reference to the class in which the method is defined. Our example, Java_Prompt_getLine, implements an instance native method. Thus the jobject parameter is a reference to the object itself.

2.4 The JNIEnv Interface Pointer

Native code accesses virtual machine functionality by calling various functions exported through the JNIEnv *interface pointer*.

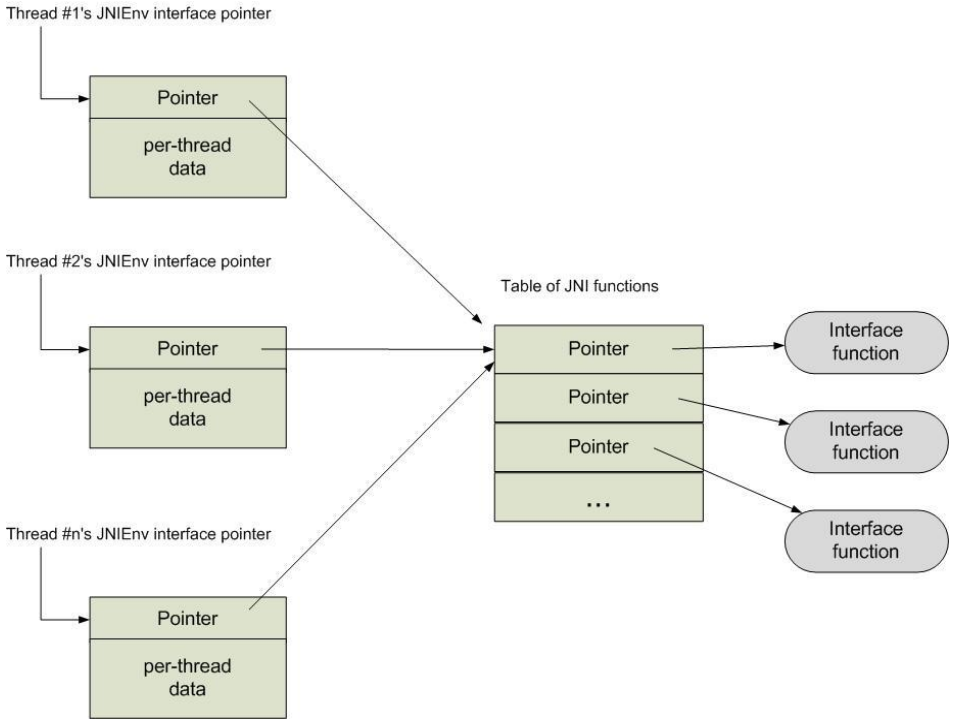


Figure 5: Thread local JNIEnv Interface Pointer

A JNIEnv interface pointer is a pointer to thread-local data, which in turn contains a pointer to a function table. Every interface function is at a predefined offset in the table. The JNIEnv interface is organized like a C++ virtual function table. Figure illustrates a set of JNIEnv interface pointers.

Functions that implement a native method receive the JNIEnv interface pointer as their first argument. The interface pointer passed to the native methods depends on the thread which calls it. The Java virtual machine passes the same interface pointer to native method functions that are called from the same thread. The calls made from different thread will have got different environment pointer. The interface pointers are thread-local but the JNI function table are indirectly referenced and shared among multiple threads.

As some of the platform does not provide efficient mechanism to support thread local data access the JNIEnv refers to the thread local structure. By passing the pointer to the thread local structure, the JNI implementation inside the virtual machine can avoid many thread-local storage access operations that it would otherwise have to perform if it was not referenced.

2.5 JNI Functions

The Java native interface provides various functions that can be called to access the Java objects and methods. To access arrays it provides get<Type>ArrayRegion .If the array contains object in Java then we need access each element

individually. Whenever we access the data it is copied from the Java heap to native region. The string must be converted from Java utf-16 to UTF-8 type. It also provides APIs to call methods in the Java form the native code. But this is rarely used in the native code as it is costly. Following tables describes various functions of JNI

JNI Function	Description	Since
Get<Type>ArrayRegion Set<Type>ArrayRegion	Copies the contents of primi-tive arrays to or from a pre allocated C buffer.	JDK1.1
Get<Type>ArrayElements Release<Type>ArrayElements	Obtains a pointer to the contents of a primitive array. May return a copy of the array.	JDK1.1
GetArrayLength	Returns the number of elements in the array.	JDK1.1
New<Type>Array	Creates an array with the given length	JDK1.1
GetPrimitiveArrayCritical	Obtains or releases a pointer to the contents of a primitive array.	Java 2 SDK1.2

Table 1: JNI Function Table 1

JNI Function	Description	Since
GetStringChars ReleaseStringChars	Obtains or releases a pointer to the contents of a string in Unicode format.	JDK1.1
GetStringUTFChars ReleaseStringUTFChars	Obtains or releases a pointer to the contents of a string in UTF-8 format. May return a copy of the string.	JDK1.1
GetStringLength	Returns the number of Unicode characters in the string.	JDK1.1
GetStringUTFLength	Returns the number of bytes needed to represent a string in the UTF-8 format.	JDK1.1
NewString	Creates a java.lang.String instance that contains the same sequence of characters as the given Unicode C string.	JDK1.1
NewStringUTF	Creates a java.lang.String	JDK1.1
GetStringCritical ReleaseStringCritical	Obtains a pointer to the contents of a string in Unicode format or a copy of the string.	Java 2 SDK1.2
GetStringRegion SetStringRegion	Copies the contents of a string to or from a preallocated C buffer in the Unicode format.	Java 2 SDK1.2

Table 2 : JNI Function Table 2

The figure below shows how a primitive array is accessed in JNI.

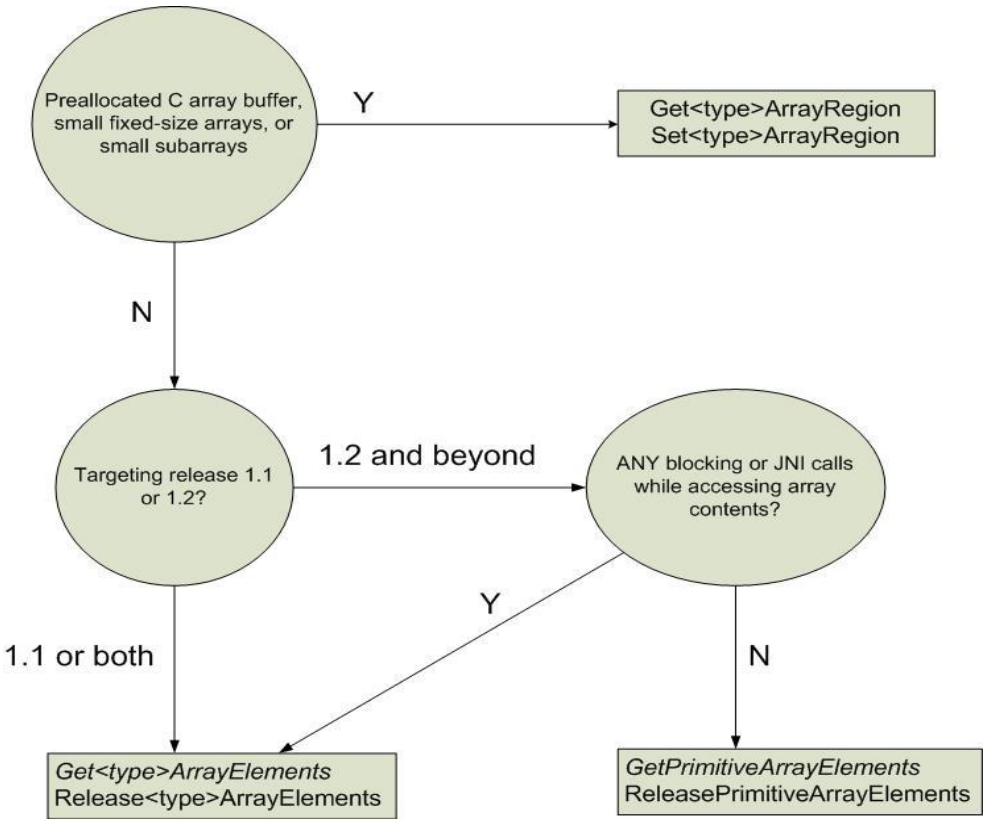


Figure 6: Primitive array access

CHAPTER 3

Android Architecture

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. Figure shows the major components of the Android operating system. Each section is described in more detail below.

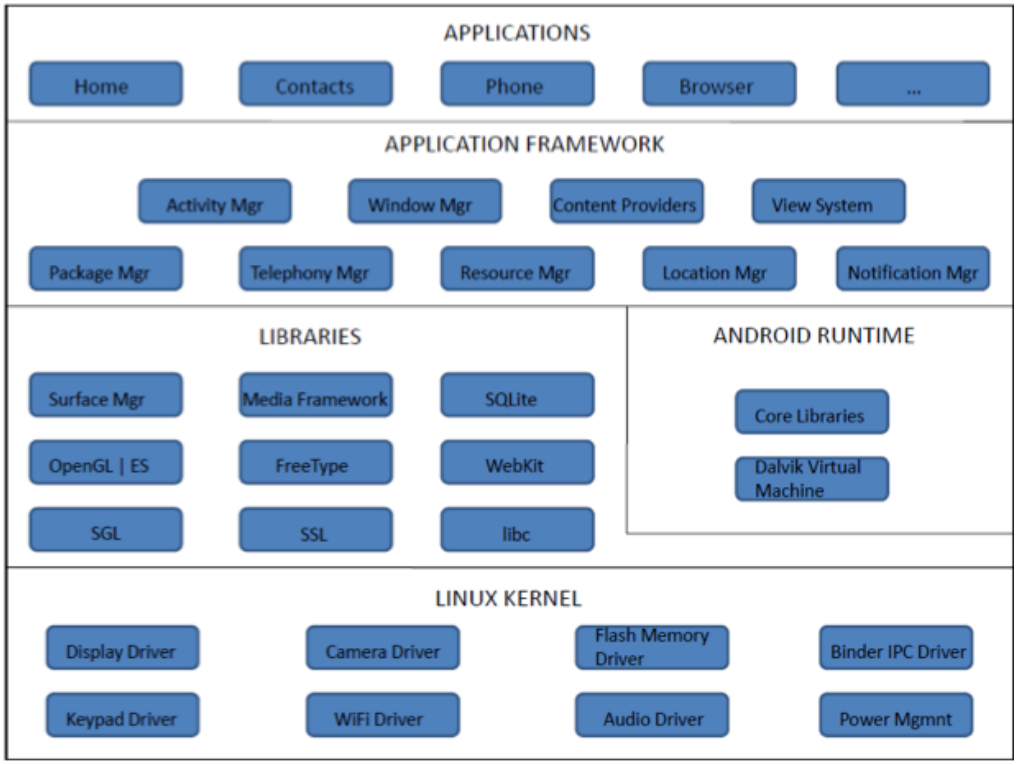


Figure 7: Major Components of Android Applications

Basic applications like contact, email, browser settings, Bluetooth etc., come with the android package. All these applications are written in the Java programming language. Many of these applications can be multi-threaded depending upon their use and interaction. Applications can be added based upon the user through Android Market.

Figure 8: Application Layer of Android

3.1 Application Framework

Application framework contains programs that manage the phone's basic functions like resource allocation, voice applications, and window allocation for applications, managing lifecycle of applications and keeping track of the phone's physical location. This layer is majorly written in the Java programming language. The API that is exposed is used by developers in their application. There is no restriction applied while accessing this Java API in the application developed by the programmer this helps in utilizing the features provided by the Android.

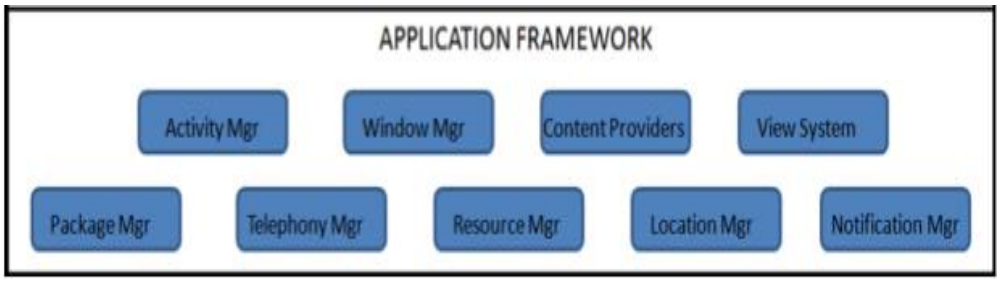


Figure 9: Application Framework of Android Libraries

Each Android component has a set of C/C++ libraries which are used by several System components. These are exposed to developers through the Android application framework. Most of the CPU related tasks and peripheral devices are

done using native C/C++ libraries (Figure 10). Some of the core libraries are:

- System C library - a tuned implementation of the standard C system library (libc), for embedded Linux-based devices
- Media Libraries - these libraries support playback and recording of many popular audio and video formats and image files.
- 3D libraries - the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.
- SQLite - a powerful and lightweight relational database engine.

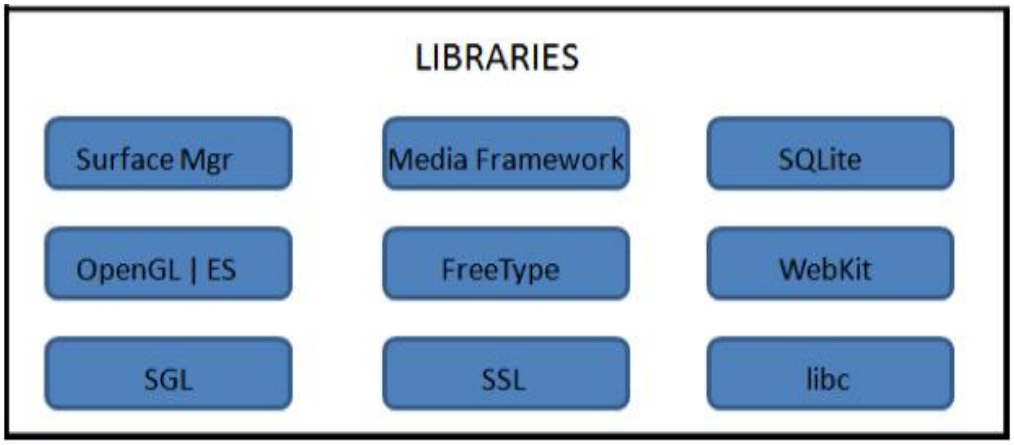


Figure 10: Libraries Layer of Android

As these libraries are for embedded systems they are optimized like fast pthread implementation using 4-byte mutex rather than the 12-byte mutex (as there may not be as many thread as compared to PC).

3.2 Android Runtime

As applications are written in Java the Android runtime environment consists of a virtual machine. Android has its own Virtual Machine called Dalvik Virtual

Machine. The generated byte code in java is converted into DEX code and an interpreter is used to convert them into assembly code. Every Android application runs in its own process, along with its own instance of the Virtual Machine

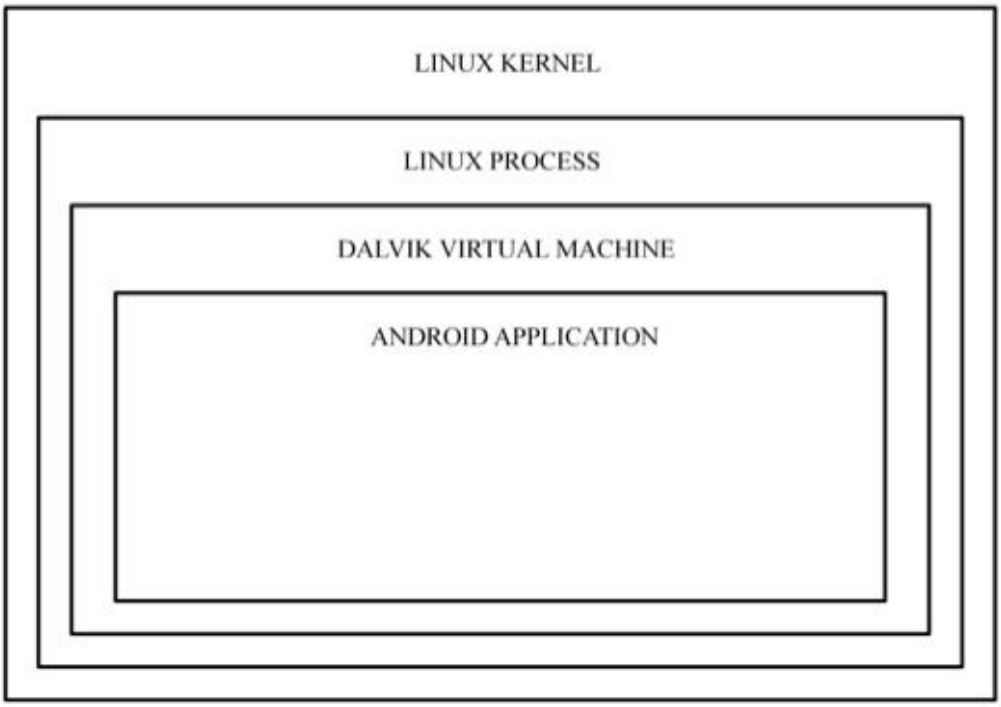


Figure 11: Layers of Android Application

Dalvik primarily is a process virtual machine. Refer Figure below. Support multiple virtual machines running concurrently. The Dalvik VM executes files has the extension (.dex) format which is optimized for minimal memory footprint. The .dex files are compiled from the .class files that are generated by Java by the "dx"tool. The VM is register-based, and runs classes compiled by a Java language compiler. The Dalvik VM is written in C and relies on the Linux kernel for core OS functionality.

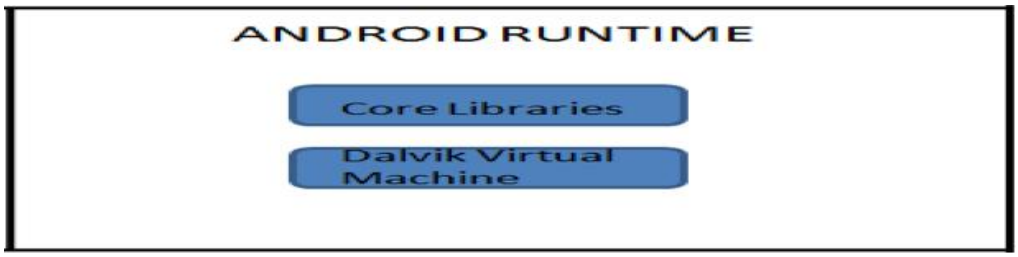


Figure 12: Android Runtime Layer

Android's kernel is based on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel acts as an abstraction layer between the hardware and the rest of the software stack. The kernel is modified so that it can cater Android specific requirements by adding drivers etc.,. Refer Figure below.

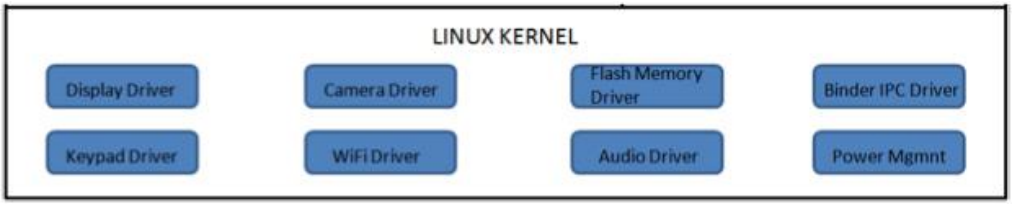


Figure 13: Linux Kernel in Android Hardware Abstraction Layer

There is an abstraction layer present in between the Linux kernel and above layers. This enables certain core default system applications and services to be replaced by third party/ custom implementations. Most mobile OEMs have the basic drivers to control their audio, video etc. Android defines this hardware abstraction layer on top of kernel and standardizes Android's interface. This hardware abstraction layer exists as a user-space C/C++ library. This probably means that Android implements a standard interface for Audio, irrespective of the technology supported by the underlying hardware, just asking implementations of

features that it needs from the particular hardware.

3.3 Android Application Architecture

Android applications are written in the Java programming language. The compiled Java code, all the necessary data and resource files of the application are bundled by the “aapt” tool into an *Android package.* This file has a .apk suffix. This file can then be used for installing the application on devices. All the code in a single .apk file is considered to be one *application*.

Android application sandbox model Android uses the process separation provided by Linux kernel as the primary means of achieving isolation against other suspicious applications. Each application runs in its own Linux process. Furthermore, each managed piece of code executes in a virtual machine (DVM). As a result each application is sand- boxed from the other applications running at any given time. All IPC is achieved via the mechanisms provided by Binder.

A second level of isolation builds upon the capability of underlying Linux to strongly isolate data/files of one user from the other. This is achieved by allocating a unique user-id to each installed application on a particular system. Android starts the process when any of the application's code needs to be executed, and shuts down the process when it's no longer needed and other applications are in need of resources. Unlike applications on most other systems, Android applications don't have a single entry point for everything in the application (no main() function, for example).

CHAPTER 4

Related work

This chapter discusses the research that has been done on the JNI. It also reviews projects that are relevant to this thesis. From the discussion above it is clear that if we decrease the time required to retrieve an object in JNI and decrease the time taken for the reflection and serialization of the data that is being transferred from the JVM to the native space we can get a better performance and save battery life of the device. Arrays are the main area of concern as huge amounts of data needs to moved to and fro. Where as if the class object is huge then time is spent on finding the attributes that we are interested in. The paper Fast Online Paper Analysis[2] helps us understand the issues of runtime pointer analysis while using reflection, dynamic loading of libraries etc., This helps us understand the issues when we use some of the JNI APIs which rely on finding and loading of classes using reflection. Walter Cazzola[3] analyses the use of reflection and its internal working. It proposes a class named SmartMethod which transforms the calls made by the use of reflection to direct calls that will be carried out similar to the standard Java method invocation. SmartInvokeC[3] tool is used to generate the stub of a class from its byte code, so to invoke a method we no longer need to use the JNI. The call is made from a C stub that is generated. To retrieve and invoke method faster the necessary information is hashed. Tamar et. Al[4] provides an memory management scheme for thread local heap. This technique determines the objects that are local and global and uses this information to avoid unnecessary

synchronization.

The following section describes some of the techniques.

4.1 JNI Bridge

One reason to use Java is that it can be ported easily on different platform as the application runs inside the virtual machine. But if the application uses native call the porting becomes difficult. That is we need to use the libraries that are specific to the platform. This paper describes the challenges and solution so that the JVM supports the native calls on different ISA. Here dynamic translators are used to translate native calls based on the underlying architecture. To handle the JNI up calls and marshaling of the data a simulated JNIEnv object in the IA-32 execution environment is used to enable 32-bit native libraries to call 64-bit function pointer. They also use marshaling tables to map 64-bit references to 32-bit references by intercepting the up calls and wrapping it with the reference and during the down call the corresponding 32-bit reference is used. To avoid the data movement when *GetPrimitiveArrayCritical* JNI API call is made the reference is directly taken from the JVM internals. The JVM-independent implementation has to resort to Java reflection to obtain this information.

4.2 Interfacing Java to the Virtual Interface Architecture

This paper explores the use of User-level network interface for the communication between the Java heap and native buffer. It describes two approaches the first approach manages the buffer between the Java heap and the

native space which requires the data to be copied while the second approach uses a Java-level buffered abstraction and allocates space outside the Java heap and this allocated space can be accessed like array in the Java. The second approach eliminates the use of copying the data but the native garbage collector has to be modified.

The first level of Javia (Javia-I)[5] manages the buffers used by VIA in native code (i.e. hides them from Java) and adds a copy on the transmission and reception paths to move the data into and out of Java arrays. Javia-I [5] can be implemented for any Java VM or system that supports a JNI-like native interface. (Javia-II) introduces a special buffer class that, coupled with special features in the garbage collector, eliminates the need for the extra copies. In Javia-II [5], the application can allocate pinned regions of memory and use these regions as Java arrays. These arrays are genuine Java objects (i.e. can be accessed directly) but are not affected by garbage collection as long as they need to remain accessible by the network interface DMA. This allows the application to manage the buffer and to send or receive Java array directly. It also describes the issues of memory management when application creates memory outside the Java heap.

4.3 Jeannie

This paper proposes a new foreign functional interface design called Jeannie[6]. Here programmers can write both the Java code and the native code in the same file. Jeannie compiles these files down to their respective JNI calls. This enables static error detection across the languages and simplifies the resource

management. It addresses the issues of JNI being unsafe as it does not require dynamic checks. By integrating and analyzing both Java and C together, the compiler can produce error messages which can prevent many a maintenance issues. The compiler is implemented using rats![6]. To access string form C in Java conversion for UTF-8 to UTF-16 is made, and vice versa is done while accessing strings from Java in C. The array region is still copied from the Java heap to C memory space when access is made, the following functions are used to gain the access _copyFromJava and _copyToJava.

4.4 Inlining Java Native Calls At Runtime

This technique inlines the native functions using JIT in java applications. The callbacks to the JNI are transformed into their equivalent lightweight operations[7]. IBM TR JIT[7] compiler is used as it supports multiple JVMs and class library implementation. The control flow for the TR JIT consists of phases for intermediate language (IL) generation, optimization and code generation. They have enhanced the inliner so that it can synthesize the opaque call to the native function. Then they have introduce a callback transformation mechanism that re-places expensive callbacks with compile-time constants and cheaper byte code equivalents, while preserving the semantics of the original source language.

4.5 Janet

Janet[8] is the Java language extension which enables convenient development of Java to native code interfaces by completely hiding the JNI layer from the user.

The source file is similar to ordinary Java source file except that it may contain embedded native code (in terms of native method implementations), and the native code can easily and directly access Java variables as Java code would. It enables efficient direct access to Java arrays from the native side. However, when the array is to be processed by external routine the array pointer has to be used. Java types are converted by Janet generally to native types having the same name. The array conversion introduces no performance reduction on platforms where appropriate Java and native types are equivalent, but it requires allocation and copying of the whole array in the case when they are different.

4.6 Native Code Profiling

This paper describes the technique used to profile native code that are part of the application. Most of the profiling tool like 'hprof' do not segregate the time spent in native code if we have this information then we can find the parts of the code that can be improved further. The paper[2] introduces a profiling tool based on JVM tool interface. The technique involves introduction of a wrapper methods for the native function prototype in Java. It has the same method name and signature as that of the native method but not a native method. This wrapper function calls the *J2N_Begin* which recodes the time stamp and other profiling information. Then it calls the native method. Upon return the wrapper calls *J2N_End* is called which records the exit timestamp. Her profiling is done statically by using a tool called ASM[2], as dynamic profiling overhead.

CHAPTER 5

Design

This chapter talks about the overview of the changes that are made to the existing JNI in android. We first need to understand the class structure of a Java class structure and how the methods and fields are represented in Java. The following section describes these representation:

5.1 Class Structure

In Dalvik virtual machine the class can be either '.class' or '.dex' extension. We are interested in the '.class' format as JNI API uses them. The class structure is defined in the Object header file. Here are the fields that are of our interest:

- Status- It is a structure of type ClassStatus through which we can know the sate of initialization like initialized, ready, loaded etc.,
- super- It holds the reference to its super class if any other wise NULL.
- Interfaces – It is a two dimensional array which contains the list of interfaces in this class.
- DirectMethodCount and directMethods– DirectMethodCount hold the count of the direct methods that are present in the class. Direct methods are static, private and init methods. DirectMethod is an array which points to the methods
- VirtualMethods and VirtualMethodCount – VirtualMethodCount holds the count of the virtual methods present in the class. VirtualMethod points to these

methods.

- Vtable - Virtual method table (vtable) is for use by "invoke-virtual".

The vtable from the superclass is copied in, and virtual methods from this class either replace those from the super or are appended.

- sfields – These are structures of type StaticFileds and contains the type of the field and its Jvalue.
- Ifields – These contains all the instance fields of this object. Also all the instance fileds that refer to objects are present in the beginning. Each instance filed object has its field type and an offset which is the offset from the object pointer.
- SourceFile – This is the name of the source file.

Every method is represented by a structure called *Method.* The structure *Method* has the following members:

- clazz – This points to the class that this method belongs to.
- MethodIndex – This contains the offset from either the vtable or the iftable of its class.
- Name – This is the method name.
- Prototype – This is the method prototype descriptor string that contains the return type and the argument type.
- Insns – this contains the actual code for the method.
- NativeFunc - This is pointer to native method. This could either be a JNI bridge function or an actual internal native function. This can be checked by

performing a null check on the insns field.

- JniArgInfo – This contains cached JNI argument and return type hints.
- InsSize – This contains the count of the number of input argument to the this method.
- OutsSize – This contains the count of the number of return arguments of this function.

The fields in the Dalvik virtual machine is represented by the structure *Field*. It contains the following members.

- Clazz - The pointer to the class it belongs to.
- Name – The name of the field.
- Signature – Its signature like "I", "[C", "Landroid/os/Debug;"
- accessFlags – The access type.

Once the JNI has found the method that needs to be executed it transfers the control to the DVM interpreter. The interpreter checks weather the function is Java function or native. If the function is Java then the byte code is interpreted into the architecture specific code.

Here is how the JNI calls are made to invoke a main method for an application :-

JNI_CreateJavaVM will be called to construct a Dalvik virtual machine.The JNI_CreateJavaVM will create everything needed to execute the .dex file, such as dynamic memory management, thread, bytecode verifier, etc. Then JavaVM and JNIEnv arguments will become the function interfaces to provide supported

functions. JNI function FindClass(JNIEnv* env, const char* name) will be called to find the class by name. Before executing a main method in Java program, its class is needed to be found by a specified name. GetStaticMethodID(JNIEnv* env, jclass jclazz, const char* name, const char* sig) is called because the Java main function must be a static method, this JNI function will be called to get the main method ID. CallStaticVoidMethod(JNIEnv* env, jclass jclazz, jmethodID methodID, …) This function will start to interpret the .dex file. This function will call the Dalvik interpreter to interpret the .dex file.

5.2 Hashing JNI fields

There are several ways to store the data and use it future so the next call made to function is faster if the programs future request can be handled based on the previous . Hashing is the technique used here to make the JNI call efficient. As JNI accesses fields in the Java domain it first needs to know the exact memory location of the field. As we have seen earlier each function in the JNI API has access to the JVM environment pointer that it is running in. This JVM environment pointer contains the pointer to the heap and the refernces to the object. It can also access the class structure of the classes that are loaded and that are present in the class path or in the jar file that is included in the class path. When we make a JNI request to access field the JNI API first checks if the class is loaded or not. Then it access the class structure and goes through the field list. If it finds the field that we are looking for it returns the fieldId to the caller. Now the caller can access this field's data using this Id. This as you can see can take a long

time. To solve this issue hashing technique is used. The hash function design is as follows. There are three different hashing techniques used here. The first one is to hash and store the class structures. The second is used to store the offset and the methodId of methods that are called through the JNI regardless of weather it is native method or Java method. The third is for the fields that are accessed by the JNI in its native domain.

The hashing bypasses the reflection calls that need to be made every time when we need to access a method or field. Now due to hashing the reflection call is made during the first access any future access will use the value obtained from the hash table. One problem with this approach is the we will not know the size of the hash table that we need to create. This issue is addressed by increasing the size by a set amount when the hash table reaches its limit. To compute the hash of the string we use the predefined function in the UtfString file.

There are always possibilities that the two stings may hash to the same key. This is called collision. To handle this probing technique is used. In this when we try to insert an entry into the hash table and we find that the slot in the hash table already full then we first check if the vale is the same as the old one if so we replace the value with new one as this is the latest value. If the value does not match then we store it in the next available free slot. This technique is chosen over the chaining for collision resolution as it uses the memory in an optimal way. Due to the use of probing while inserting the value in the hash table the retrieval takes longer. We need not address the synchronization issue as JNI API is

executed by a single thread and there is only one thread accessing the JVM environment variable.

There are three hash table defined one each for to hold the class structure, field memory location and methodIds. The following are the names of the hash map

- fieldEntryJni
- refEntryTable
- methEntryTable

These hash tables are updated when the call to FindClass, getMethodId and getFieldId is called by the JNI API. These hash tables are initialized when the new JNI environment variable is created when *dvmCreateJNIEnv* is called during API invocation.

5.3 Find Class

The find class in the JNI is used to load the class given the fully qualified class name in the JVM pointed by the JNI environment object. The find class takes in two argument one the JNI environment object and the class name. The findClass first make a check by calling *dvmGetCurrentJNIMethod,* this method check if the current thread is executing a native method if so it returns the method by inspecting the interp stack. Then we get the class descriptor form the class name. The class name is surrounded by 'L' and ';'. Then we load the class from its class loader and add the local reference in the reference table. This is a very important step if the class does not have a reference then this class cannot be accessed in the

native stack. Then we return the reference to the caller who can use it to access fields and methods.

If the class was already loaded then we could reuse this instance of the class to get the fieldIds and methodIds. To accomplish this we store the reference of the class in the hash table *refEntryTable* in the JNI environment variable. Every time the findClass is called we check if the class already loaded by looking up in the hash map. If not we proceed as usual and load the class. After loading the class we add this to the hash map. If the class we are looking for is present in the hash map we validate its reference as it may be garbage collected if the reference is valid then we add it to the local reference table so that the class reference is not garbage collected. Then we return this instance to the caller. This decreases the time taken to load the class. The key that we use here is the class descriptor rather than the class name passed. This is because the two classes can have the same name but the descriptor are different.

The below flow diagram for the findClass method

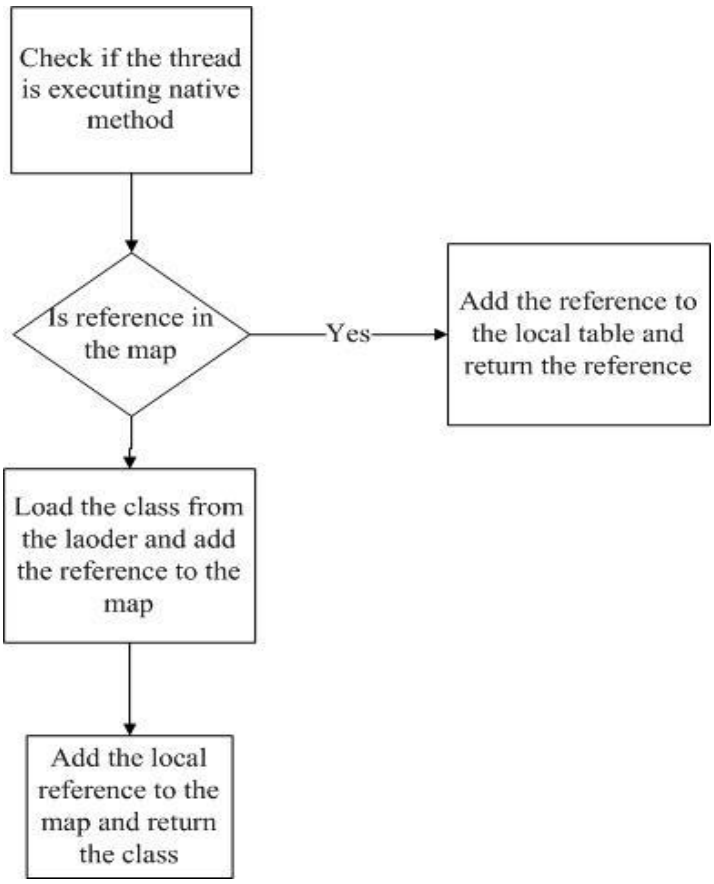


Figure 14: Flow of findClass

The method findClass is used to load the applications main method hence optimizing this will reduce the execution time of the application. As long as the local reference of the class is present the class is not unloaded hence we can reuse it. These are methods for the hash table access

- hashcmpRefEntryTableStr
- hashcmpRefEntryTable
- findRefEntryTableEntry
- addRefEntryTableEntry

5.4 Get Field

The getFieldId is used to get the instance field give the field name, its signature, and the Java class. To access any field in the Java object we need to use this function. It returns the fieldId in the class structure through which we can get the offset from the object pointer. The Java class structure is obtained from the previous findClass. As we have the reference in the local reference table of the JNI we need to get the reference of the actual class. This is accomplished by the function *dvmDecodeIndirectRef*. The we check if the class is initialized. Then we find the filedId as follows

- First we search the class object for the given field name and signature.
- This is done by walking through the ifields of the class that is provided. If found the filedId is returned.
- If it is not found in the class then we see if there is any super class present. If yes then we scan through the ifields of the super class. If it is found here then we return this fieldId.
- If the field is not found then we throw an exception.

As we can see this procedure of scanning through the list can take a lot of time. Once we find the fieldId we can add it to the hash. But this is tricky as multiple class have the same field name and signature. To make the key unique there is a combination of the field name, field signature and class descriptor. Hence this is unique. Here we also need to check if the class is loaded or not when we return the fieldID from the hash table. If the class that the field belongs

is not loaded the further use of the Id will generate exception. This can be taken care by looking into the hash table of findClass as this will take only O(1). If the class is not loaded then we assert an exception similar to the actual flow.

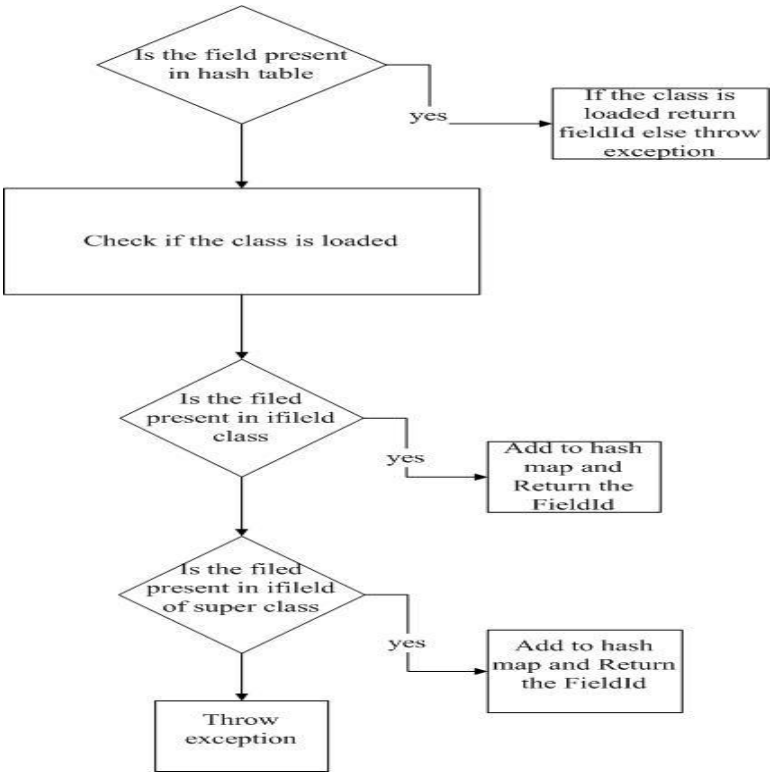


Figure 15: GetField workflow

The above diagram shows the function flow.These are methods for the hash table access

- hashcmpFieldEntryTableStr
- hashcmpFieldEntryTable
- findFieldEntryTableEntry
- addFieldEntryTableEntry

5.5 Get Method

The GetMethodId is used to find the methodId for a given method name, signature and class. Similar to the GetFieldId to get the methodId we first need to load the class by calling findClass. Form this methodId we can get the offset from the class pointer and invoke the method. The following is performed by this method

- It first checks if the class is loaded and initialized. If not an exception is thrown.
- It then goes through the list vtable to check if the method and the signature is present if present it checks if the method is static. If not then it returns this methodId.
- If the it is not a virtual function then it goes through the directMethods list to check if it is present in this. If present it checks if the method is static or not. If static then it throws exception else returns the methodId.
- The method's class may not be the same as class that is supplied , but if it isn't this must be a virtual method and the class must be a superclass Hence we initialize the super class.

We add the method Id to the hash table represented by *methEntryTable*. When a call is made first we check if the method is present in the hash table if so we check if the class and its super class is initialized or not by checking the hash table for the refEntryTable. If the class is not initialized we initialize it add it to the refEntryTable and also add it to the local reference. The key used here is the class

descriptor, the method name and the signature. Hence this key is unique.

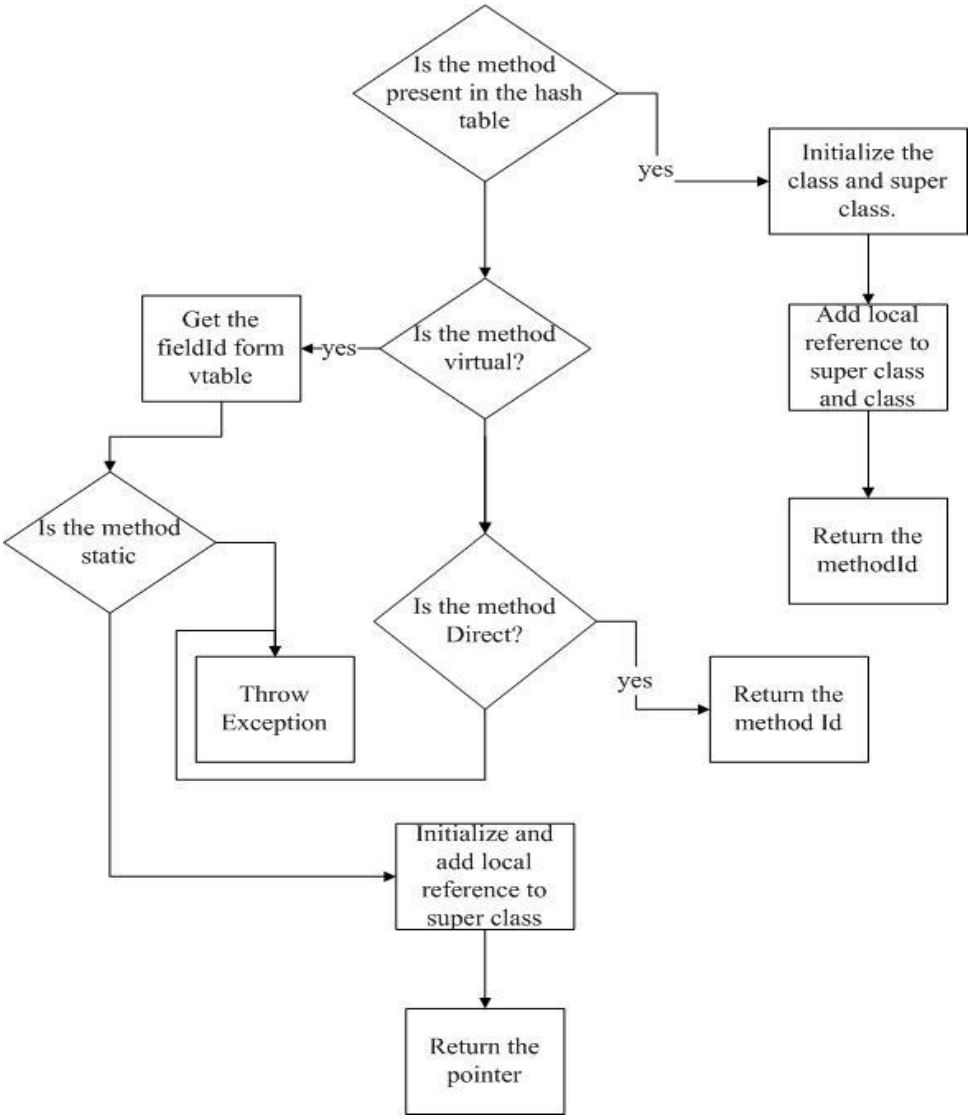


Figure 16: Get Method

The above digram shows the flow for the GetMethodId function. These are methods for the hash table access

- hashcmpMethEntryTableStr
- hashcmpMethEntryTable
- findFieldMethTableEntry

- addFieldMethTableEntry

5.6 Pinning object

To access objects we need to go through the reflection and serialization in the JNI. If we pin the memory location and obtain the memory location in the native space we can access its field by knowing the offset. Pinning not only allows us access the object but makes sure that the memory location of the object does not change. It is also important to unpin the object once we are done using it as these can be picked up by the garbage collector when in Java program it is assigned to a new reference. For this two new functions are provided *pinObject* and *unpinObject* . The pinObject method adds an entry to the global DVM pin reference table. The pin reference table is a global table and has a limit to the number of reference it can hold. If we cannot add it an exception is thrown. First we add this object to the global reference table so that while pinning we can check if the address is a valid object. We also increase the global count if it already exists and also make sure that it is pinned only once. If it is pinned more than once then the release has not been called.

In the unpinObject function we remove the reference from the global DVM reference table. We also remove the global reference that we added. To access the global DVM reference table we need to obtain mutex lock over the table. Care should be taken while accessing the attributes an changing it as the memory that is obtained has no restriction on it. If we override the method table or any other vital data the object will become corrupt. Also currently only a few objects can be

pinned as the global table has limited count.

The figure below show the approach of pinning the object

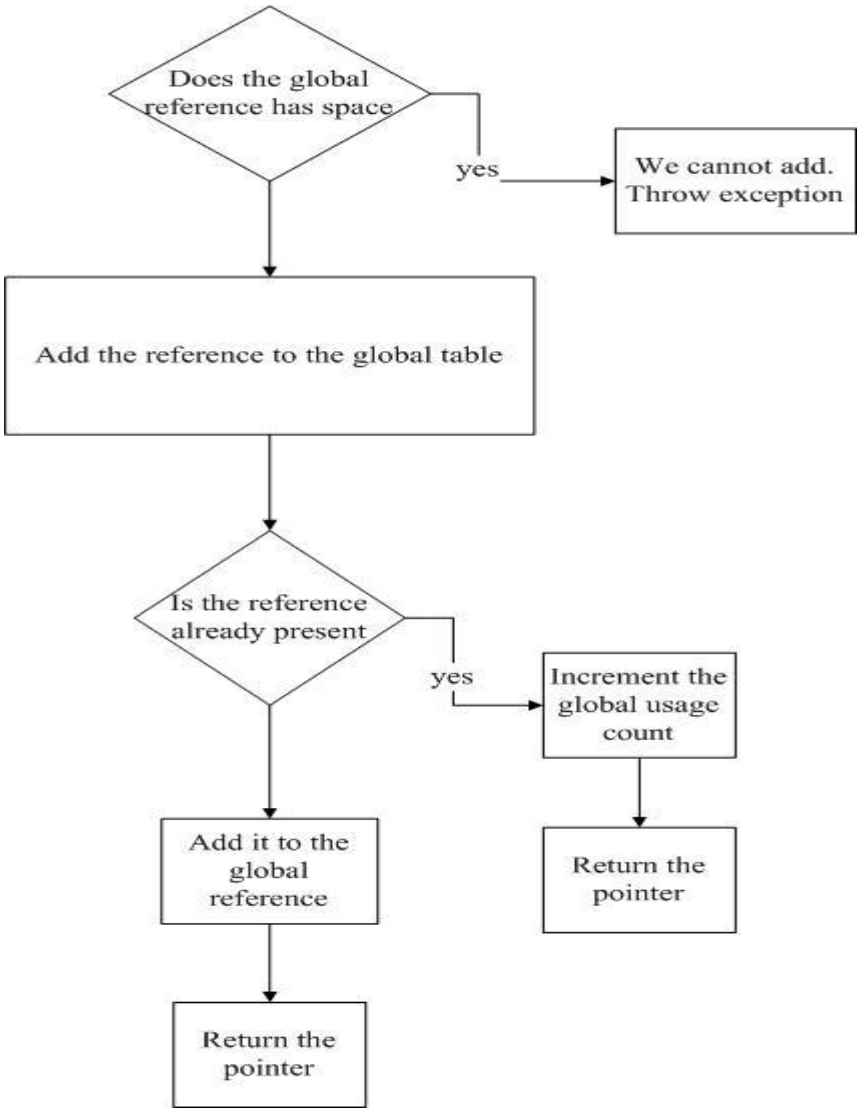


Figure 17: Pinning object

CHAPTER 6

Implementation

The following section gives the implementation details of the hash functions used and the changes made to JNI API.

6.1 Implementation of hash function

The hash function has the following functions

HashTable dvmHashTableCreate(size_t initialSize, HashFreeFunc freeFunc)*

This function is used to initialize the hash table. The free function depends on the value that we are storing. In our approach three different functions have been used as there are hash tables for findClass, GetFieldId and GetMethodId. The basic operation that the free function does is that it removes the local reference that we have added for the classes that are related to the value that we are storing.

void dvmHashTableLookup(HashTable* pHashTable, u4 itemHash, void* item, HashCompareFunc cmpFunc, bool doAdd)*

This function has dual purpose that is it servers the purpose of look up when the doAdd flag is set to false and it inserts the <key, value> pair when the flag is true. It also takes in the compare function. Our approach has used two different compare functions one while adding entry to the hash table and the other while retrieving the value from the hash table. Since while adding we need to compare only the hash value of the key and the collision is taken care by probing the compare function is simple. During the retrieval of the value we need to perform

checks based the data that we are retrieving. The hash is computed using the UT8 string which is the key.

void dvmHashTableLock(HashTable pHashTable)*

This function is used to lock the table so that no two threads change the data simultaneously. Each hash table has its own mutex lock. This function grabs the lock.

void dvmHashTableUnlock(HashTable pHashTable)*

This function is used to release the lock on the hash table once the operations on it completed.

static bool resizeHash(HashTable pHashTable, int newSize)*

When we initialize the hash map the size of the map is set to 50 entries. But during the program execution the size may increase. This function allows us to resize the hash table. To perform this operation we need to remove all the elements and reenter them into this bigger hash table. To take care of the null values the use of tombstone constant is used. This is an expensive task as all the values needs to be reentered into the table. To perform the resize operation the table must be 75% full. This is the threshold value that we have used for the hash tables.

bool dvmHashTableRemove(HashTable pHashTable, u4 itemHash, void* item)*

This function removes an item from the hash table with the given hash. The hash function that has be used is as follows as the keys are strings and are of UTF8 format we use the below formulae to calculate the hash

hash = hash *31 + value of character

This hash is computed for the whole length of the string. Here we presume that add and look up happen more frequently than the remove. Whenever remove function is called there should be an explicit call to the free function as well as the remove function does not invoke it internally. If the free function is not called then there is high chances that we will be running out of space in the local reference table of the JNI environment. Similarly the free function also handles the global references that we have created for the values that we store.

6.2 Implementation of GetFieldId

As discussed in the earlier section the we are using an hash map int the JNI environment variable that is passed with the JNI API call. To insert into the hash table the function *addFieldEntryTableEntry* is used. This function takes in the hash table, the class pointer, the field name and its signature. The key is combination of class->descriptor plus the name plus signature. This key is hashed using the technique described previously. While adding if the slot is already occupied the we retrieve the vale and the following check is made

fieldId->clazz->descriptor is compared with class descriptor

fieldId->name is compared with the field name

filedId->signature is compared with field signature

If all of the following matches we replace this with the new entry by removing this value. If not probing is used to find the next empty position. During the retrieval of the value we do the following check

fieldId->clazz->descriptor +

fieldId->name +

filedId->signature are concatenated in the above order and is compared to the key which is the input.

If they are equal then we check if the local reference is valid.

If it is valid return the value else

remove the <key, value> from the hash table.

While adding the entry we check whether the class is initialized by invoking the *dvmIsClassInitializing*. The same is done while retrieving the class. Also checks are made to see if the local references are still valid otherwise we add the local reference to this class so that no exception occur in the future when the field values are accessed in the native domain.

6.3 Implementation of GetMethodId

The GetMethodId is a bit more complicated than the GetFieldId. Here the method can either be a direct method or a interface method. If it is an inter face method then we need to initialize the super class and add local reference to it. Here we are use *addMethEntryTableEntry* to insert it into the hash table.

The key here is the class descriptor, method name and its prototype. Here the prototype contains the return and the argument list. This combination will be unique for a class.The comparison that we make while adding is the following

meth->clazz->descriptor is compared with class descriptor.

meth->name is compared with name

meth->prototype is compared with the sig

Comparing the prototype and signature is handled by the method *dvmCompareNameProtoAndMethod.* If these match the we do not replace the value but if the method is virtual the we check if the super class is loaded. While retrieving the value when it is present in the hash table we check the following

check if the method object is valid

check is made to see if the class is initialized if not remove the object and throw exception

check if the super class is initialized if not remove the object and throw exception

If everything is fine add the local reference to the classes.

If we find that the method is static then we throw an exception as static methods cannot be accessed by the GetMethodId. Also check is made to see if there is code to execute in the method while returning the method Id so that we do not catch any abstract method.

6.4 Implementation of pinObject and unpinObject

The hard part of accessing an object is that we need to do find class first then find the field offset and the get the value by invoking the get method for the object in JNI. This is because the garbage collector may move the object however if we pin the object we are guaranteed that the object will be in same memory location. This is very helpful as we can access array of objects if we know the starting location and size of the object instead of calling getObjectArrayElement.

The following is part of the code which adds the object to the global dvm

table.

dvmAddToReferenceTable(&gDvm.jniPinRefTable, (Object)Obj)*

This assures us that once we add this object to this reference table the garbage collector will no longer move or reclaim this memory location. Once we have finished we need to call the unpinObject method so that the garbage collector can reclaim this memory and/or move the object.

6.5 Implementation of FindClass

In findClass when the class is requested we first get the class descriptor from the method *dvmNameToDescriptor.*The we hash this descriptor and find out whether this is present in the table if so we add local reference so as to make sure that this is not garbage collected. Then return this reference. While retrieving we do the following

class->descriptor is compared with descriptor that got through *dvmNameToDescriptor.* If there is a match then we return the reference and add it to the local reference table.

Below is the code snippet on how the comparison takes place

```
ClassObject* clazz = (ClassObject*) ventry;
        ClassObject* clazz1 = (ClassObject*) vnewEntry;
        LOGI("desc val: %s, %s ",(const char*)clazz1->descriptor,(const
char*)clazz->descriptor);
        if(clazz!=NULL && clazz1!=NULL){
                return strcmp((const char*)clazz1->descriptor,
```

```
        (const char*)clazz->descriptor);
}
return 0;
```

CHAPTER 7

Evaluation

This section compares the JNI interface in the Dalvik virtual machine and the JNI with the proposed changes. To evaluate the execution time in the JNI we record the system clock in microseconds. This is accomplished by the following function is used to get the CPU time.

```
static inline u8 getClock()
{
#if defined(HAVE_POSIX_CLOCKS)
    struct timespec tm;
    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tm);
    if (!(tm.tv_nsec >= 0 && tm.tv_nsec < 1*1000*1000*1000)) {
        LOGE("bad nsec: %ld\n", tm.tv_nsec);
        dvmAbort();
    }
   return tm.tv_sec * 1000000LL + tm.tv_nsec / 1000;
#else
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return tv.tv_sec * 1000000LL + tv.tv_usec;
#endif
}
```

The time is logged into the system log file using the following command

LOGI(...)

This is a predefined function in the Dalvik virtual machine which internally uses printf command to the output the sting into the system log buffer. This is then flushed to Android Debug Bridge console. The next section gives a brief background on Android Debug Bridge (adb).

7.1 Android Debug Bridge

The adb is a tool which allows us to manage the state of the emulator instance. It is a client server program with following three components

- Client – This runs on the development machine. We can communicate to the device by issuing various *adb* commands.
- Server – This is a background process that runs on the development machine. This manages the communication between the clients and the adb thread that runs on the emulator/device.
- Daemon – This is the background process that runs on the device or emulator.

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process.

To view the logs we issue the following command once we connect the emulator to the adb server

$ adb logcat

This logging system provides a mechanism for collecting and viewing system

debug output. Logs from various applications and portions of the system are collected in a series of circular buffers, which then can be viewed and filtered by this command. Each log message is associated with a tag and a priority. These are the different priority associated with the log

- V- verbose(lowest)
- D – Debug
- I – Info
- W – Warning
- E – Error
- F – Fatal
- S – Silent

To filter the log the following command is used

adb logcat *tag*:I *AppName*:D *:*loglevel*

This displays only the log statements that we are interested in. We also tag each of these log messages with their corresponding processId and threadId.

- Brief - Display priority/tag and PID of originating process (the default format).
- Process - Display PID only.
- Tag - Display the priority/tag only.
- Thread - Display process:thread and priority/tag only.
- Raw - Display the raw log message, with no other metadata fields.
- Time - Display the date, invocation time, priority/tag, and PID of the

originating process.

- Long - Display all metadata fields and separate messages with a blank lines.

We have used the display option of *thread* in our logs. The log statements have been put in the JNI_ENTRY and JNI_EXIT macro. These macros are invoked each time there is a call to JNI API. In each of the JNI API we log the method name so that we can identify the time taken in each of these calls. The next section will explain the results and comparison of the methods that have been changed.

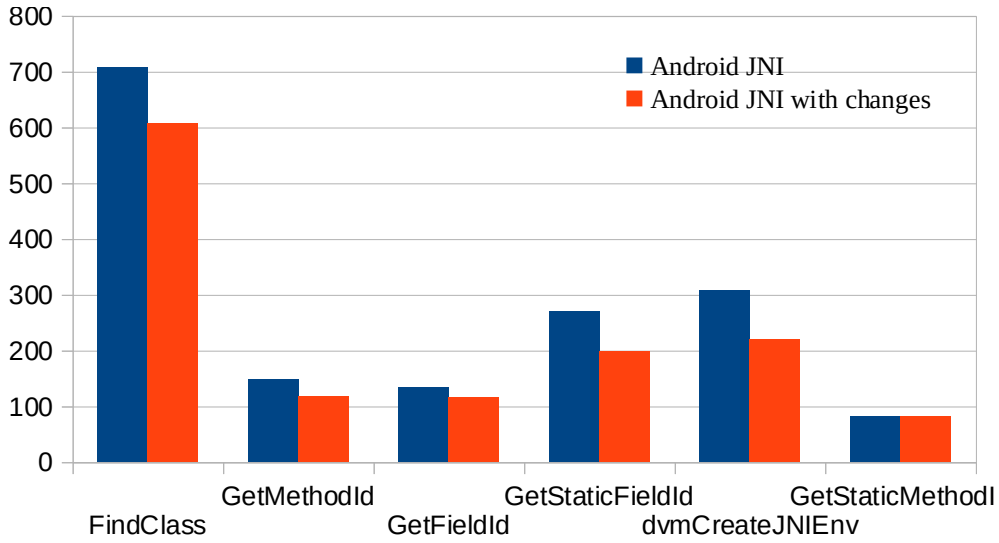
7.2 Profiling results

As explained earlier we are using logging provided by the Android Operating System and the system clock to profile the JNI calls. Here we profile the JNI calls that we have changed. We ran the application lunar lander and recoded the time in each of the JNI calls by logging the system time. Then we took the average of the calls. The below table shows the time.

Time in microseconds	FindClass	getMethodID	getFiledID	getStaticFiedlID	dvmCreateJNIEnv	GetStaticMethodID
JNI without changes	708.18	147.84	133.18	269.4	308	80.7
JNI with changes	606.38	117.05	114.9	198.4	281.63	82.16

Table 3: JNI method profile

This is due to the fact that the subsequent calls to these methods need not go through the reflection of the object to get the field or class. The calls still require



to initialize the object so that the reference is present int JVMs local reference.

Figure 18: JNI method Profiling

The below table gives the timings of the heap sort. First we implemented the heap sort in Java and recorded the execution time. Then the heap sort was ran using JNI and the sort was implemented in the native domain. Then we ran the same code and the data on the improved JNI. The table below shows the result for the various input size.

Array Size	Dalvik Java (ms)	JNI modified (ms)	Android JNI (ms)
500	34.38	30	30
1000	40	30	32.81
2000	61.45	32.77	36.53
3000	63.54	31.42	40
4000	68.82	37.64	43.07
5000	76.74	43.51	48.37
6000	85.63	50.04	55.53

Table 4: Heap sort running time

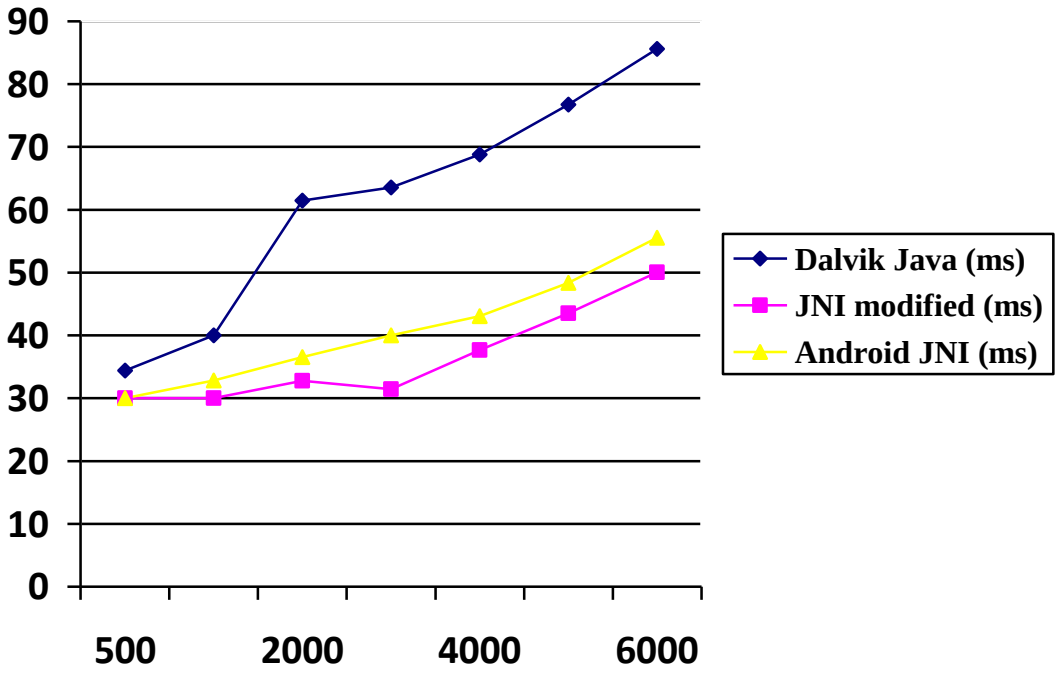


Figure 19: Heap sort comparison graph

As we can see from the graph the execution time in the modified JNI decreases as the input size increases. This is due to the fact that there is more data to be moved from the Java heap to the native space.

Next we run record the application time in Java, native calls and Operating system. Form this we can see the amount to time that is saved due to the techniques proposed. For this we use *gprof* to profile the emulator. Here we need to address the issue of scheduling that is done by the kernel when multiple applications are running. To tackle this issue only one application is loaded into the emulator so that all the resources are used by this application. The application

that we choose was *jetboy.* The application changes the music based on the events that occur in the game. It uses the JET library that is provided by android to play the music file. OpenGL is used to render the graphic contents that is used in the game. Both these use JNI by the means of NDK library. The below graph depicts the running time of the application with the JNI API provided by the android. Here we do not consider the time taken by the application to start. We run the application for 2 seconds.

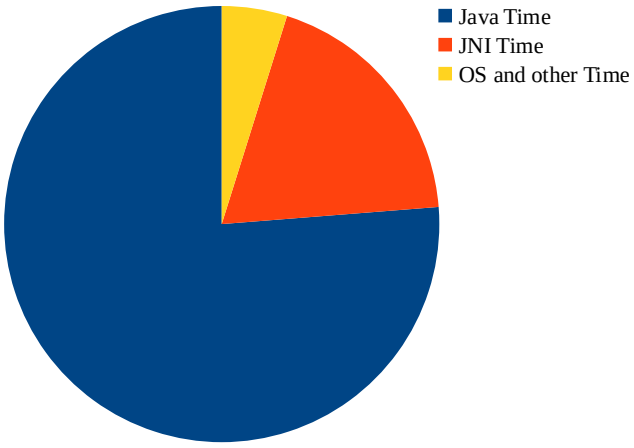


Figure 20: Execution time of JetBoy

There were ***1173*** calls made to the JNI interface during the program execution . Here we captured the time in the JNI API using the system time stamp. To find the number of JNI calls that were made were recorded by a static variable declared in the JNIEnv variable. Once the JNI destroy was called we printed it to the log. Then we summed up all the count. The table below shows the execution time of the JetBoy.

Environments	Time in mS
Java	1.49
JNI	0.37
OS and other	0.1

Table 5: Execution time for JetBoy

The same experiment was conducted using the modified JNI. Though the number of calls remained almost same there was an improvement in the JNI time as the table below shows. This is mainly due to the calls made to the findClass and getFieldId API methods. The total number of calls that were made to the JNI were ***1184***

Environments	Time in mS
Java	1.48
JNI	0.31
OS and other	0.1

Table 6: Running Time for JetBoy with modified JNI

Time in microseconds	FindClass	getMethodID	getFiledID	getStaticFiedlID
JNI without changes	638.43	206.290	133.18	362.346
JNI with changes	572.37	168.5	92.9	294.35

Table 7: JNI modified method profile for JetBoy

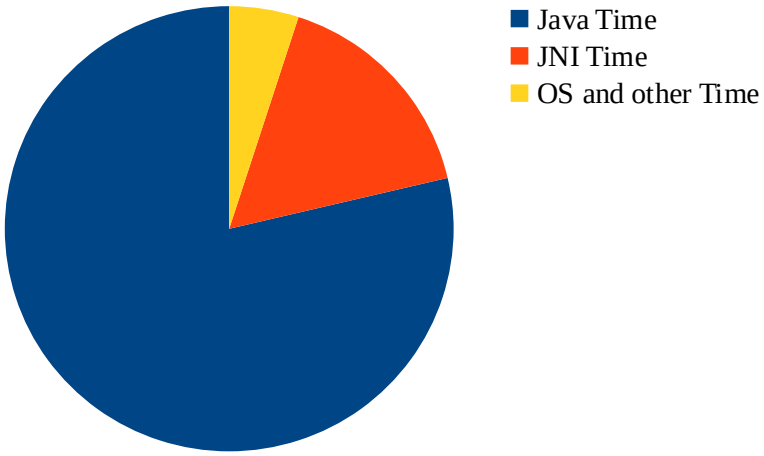


Figure 21: Execution time for the Modified JNI JetBoy

The same was done for Lunar Lander application. This application demonstrates the drawing resources and animation in Android. The graph below shows the execution time for the application with unmodified JNI.

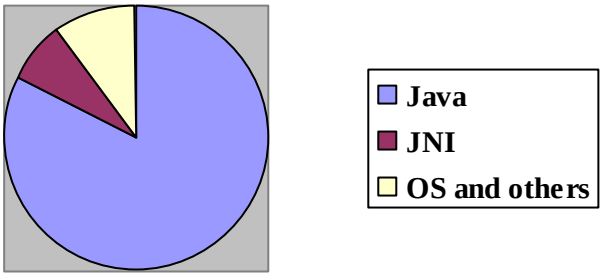


Figure 22: Execution time for Lunar Lander with Android JNI

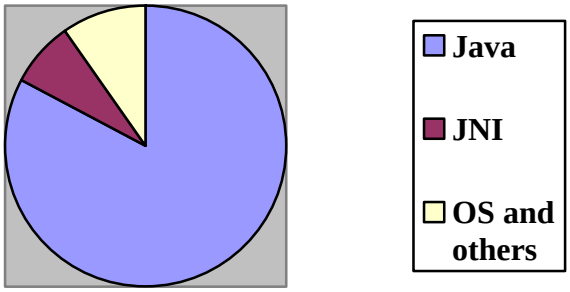


Figure 23: Execution time for Lunar Lander with modified JNI

To analyze the time taken to access the reference object we wrote a program to access the object which has a reference object as its member. The following were the results for the JNI modified and unmodified JNI.

Time in micro seconds	Reference access
Android JNI	1307.742
Modified Android JNI	1215.64

Table 8: Reference object access time.

While accessing the reference object we need to have the class structure of both the parent object and the member object. As we remove this step by caching the object structure this computation is saved and we see that the modified JNI is faster

CHAPTER 8

Conclusion

JNI has always been an important part of the Java virtual machine. It is widely used in embedded application as they are architecture specific libraries in native code (C/C++) which can improve the performance of the application. Our technique reduces the over head of reflection and serialization that are used while accessing the objects by the JNI. The pinning of objects helps the programmer to reference the data through its memory location rather than copying the object into the native space. There is an performance bonus of 5%-10% achieved using our technique. As we have seen by inlining the JNI calls[7] there can be a gain in the performance but these are not applied to the android JNI as it is new to the market. The Janet[8] provides the programmer an easy way to integrate the JNI and Java with type safe and static error checking.

There is not much research done in the JNI pertaining to android. This thesis shows that JNI performance can be improved by reducing the overhead of synchronization and by caching the class information for future use.

CHAPTER 9

Future Work

This thesis provides a technique that improves the running time of android application that use JNI. The technique uses pinning of objects so that it can be accessed through reference rather than copying the data. It also shows that if we cache the information of the class and fields there is a an improvement in the performance. There can be performance increase by inlining and use of JIT in the JVM. Currently we are not looking into the stack when the transfer is being made to the JNI and back. If we could reduce this over head then we make inexpensive calls to the JNI and use it frequently. Also the profiling of the JNI is done using the execution time of the application but we can gain a deeper insight once we inspect the instruction profile in the JNI. This will provide further areas of improvement.

As android is new to the market there is no benchmark application that can be used to profile the platform.

REFERENCES

[1] Sangchul Lee and Jae Wook Jeon "Evaluating Performance of Android Platform Using Native C for Embedded Systems" Control Automation and Systems (ICCAS), 2010 International Conference pages 1160 - 1163.

[2] Walter Binder, Jarle Hulaas and Philippe Moret "A Quantitative Evaluation of the Contribution of Native Code to Java Workloads" Workload Characterization, 2006 IEEE International Symposium pages 201-209.

[3] Walter Cazzola "SmartMethod: an Efficient Replacement for Method" In SAC'04, pages 1305 - 1309, Nicosia,Cyprus, Mar. 2004. ACM Press

[4] Tamar Domani and Gal Goldshtein "Thread-Local Heaps for Java" ISMM '02 Proceedings of the 3rd international symposium on Memory management pages.

[5] Chi-Chao Chang and Thorsten von Eicken "Interfacing Java to the Virtual Interface Architecture" JAVA '99 Proceedings of the ACM 1999 conference on Java Grande pages 51-57.

[6] Martin Hirzel and Robert Grimm "Jeannie: Granting Java Native Interface Developers Their Wishes" OOPSLA '07 Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications pages 19 - 38

[7] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents and Kevin Stoodley "Inlining Java Native Calls At Runtime" In VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pages 121--131, New York, NY, USA, 2005. ACM Press.

[8] MarianBubak, DawidKurzyniec, and Piotr Luszczek "Creating Java to Native Code Interfaces with Janet Extension" In Proc. SGI Users's Conference, pp. 283--294, Oct. 2000.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Addison-Wesley, second edition, June 2000.

[10] S. Liang. The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, June 1999.

[11] Sun Microsystems.Integrating native methods into Java programs. http://java.sun.com/docs/books/ tutorialNB/download/tut-native1dot0.zip, May 1998.

[12] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang.” Safe Java native interface”. In Proc. 2006 IEEE International Symposium on Secure Software Engineering, pp. 97–106, Mar. 2006.

[13] M.Bubak, D.Kurzyniec, andP.Luszczek “CreatingJavatonativecodeinterfaces with Janet extension”. In M. Bubak, J. Mo´scinski, and M. Noga, editors, Proceedings of the First Worldwide SGI Users’ Conference, pages 283–294, Cracow, Poland, October 11-14 2000. ACC-CYFRONET.

[14] Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/index.html.

[15] Oracle, "Java Virtual Machine Profiler Interface (JVMPI)."

[16] F. Y. Tim Lindholm, "The JavaTM Virtual Machine Specification," 1999

[17] D. Bornstein, "Dalvik VM Internals," 2008.

[18] P. Brady, "Anatomy & Physiology of an Android," 2008. http://www.oracle.com/technetwork/java/javame

[19] www.wikipedia.org

[20] Damianos Gavalas and Daphne Economou “Development Platforms for Mobile Applications” Software, IEEE Issue Jan.-Feb. 2011 Volume 28 page 77

[21] J. Andrews “Interfacing Java with native code – performance limits.” http://www.str.com.au/jnibench/